AsmetaL - A user guide

February 23, 2011

This user's guide wants to be an introduction to the AsmetaL language and to show the problems a user could find when he starts writing code in this language. A prerequisite for the lecture of the text is the knowledge of ASM theory [1]. For a complete description of AsmetaL syntax you can refer to the ASMETA web site [2] and in particularly to the guide of notation [3]. In the text, sometimes, we will use some short examples to underline some concepts; we will also use the complete models of the ATM and LIFT specifications: because of their lengths they are reported at the end of the text (sections 8.1 and 8.2).

Contents

1 ASM structure **2**

2 Domain and function declaration **3**

3 Body and main rule **5**
- 3.1 Static domains, variables and functions initialization 5
- 3.2 Derived functions 6
- 3.3 Rules 6
- 3.4 Main rule 7

4 Initialization **7**

5 Monitored function **8**
- 5.1 At what state the monitored functions values belong? 8
- 5.2 How to use monitored functions 9
- 5.3 How to identify monitored functions 10

6 Invariants **10**
- 6.1 Invariants declaration 11
- 6.2 Invariants violation 11

7 Inconsistent updates **12**

8 Examples **15**
8.1 ATM specification 15
8.2 LIFT specification 19

1 ASM structure

An ASM model is structured into four sections: an *header*, a *body*, a *main rule* and an *initialization*. In code 1 is shown an ASM model for the factorial calculation; let's examine it to understand the structure of an ASM model.

```
asm factorial

import ../STDL/StandardLibrary

signature:
    dynamic monitored value: Integer
    dynamic controlled index: Integer
    dynamic controlled factorial: Integer
    dynamic controlled outMess: Any
    derived continue: Integer -> Boolean

definitions:

    function continue($i in Integer) =
        $i>1

    macro rule r_factorial =
        if(continue(index)) then
            seq
                factorial := factorial*index
                index := index-1
            endseq
        endif

    main rule r_Main =
        seq
            if(index=1) then
                if(value>0) then
                    par
                        index := value
                        factorial := 1
                        outMess := "Executing the factorial"
                    endpar
                else
                    outMess := "Insert a value greater than zero"
                endif
            endif
            r_factorial[]
        endseq


default init s0:
    function index = 1
```

Code 1: Example of an ASM model: Factorial

At the beginnig of the file we must insert the keyword *asm* followed by the name of the model, that must be equal to the file name (case sensitive).

If the ASM is an asynchronous multi-agent, the keyword *asyncr* must be inserted before the keyword *asm*; by default an ASM is considered a synchronous multi-agent.

In the header section, we can import external modules in the following way:

```
import extModule
```

where *import* is a keyword and *extModule* is the relative path of the module we want to import. In our example we have imported the module *StandardLibrary* that contains many useful functions and that is nedeed by almost all non-trivial ASM models. A copy of the StandardLibrary can be downloaded from [5].
Then, after the keyword *signature*, we can define the signature of domains and functions (see section 2).

In the body section (see section 3), we must insert (in this order) the implementation of static concrete domain, of derived and static functions and of the rules of our model.

After the body section we can insert the main rule (see section 3); if we are writing an ASM module[1], the main rule must be omitted.

At the end, in the initialization section (see section 4), we can insert the initialization of controlled functions.

2 Domain and function declaration

In code 1 we can see that each function signature reports, after the colon, the codomain of the function.

Let's see how to use and declare domains in an ASM model.
There are several types of domains. First of all, we can choose from some basic type domains: Complex, Real, Integer, Natural, String, Char, Boolean, Rule, and Undef[2]. For example, in the ATM example (section 8.1) the function *moneyLeft* has been declared as an Integer.

Moreover we can define an abstract type domain that can be useful to represents entities of real word; in example 8.1 we have defined an abstract domain *NumCard*. The static functions *card1*, *card2* and *card3* has been declared as *NumCard*. Through the use of an abstract domain we do not have to worry about the representation of the credit card. Maybe, in a refinement step, we could change the domain and decide to identify the credit cards with integers or strings.

Sometimes could be useful to have a function that can assume values of different types: in this case we can define the function over an *anydomain*. In the ATM example the function *outMess* is used to show messages to the user; these messages can contain both strings and numbers: for this reason the function has been associated to the *Any* domain (is an *anydomain* defined in the Standard Library).

[1] An ASM module is an ASM without a main rule and without a characterization of the set of initial states. In an ASM module, before the name of the model we must insert the keyword *module* (and not *asm*).

[2] We can see that the first letter of the name of a domain must be upper-case.

In the same example we can see another type of domain: the enum domain. The enum domain is useful to define an enumeration of elements; in the ATM example the domain *Service* contains the three services supplied by the ATM (“BALANCE “, “WITHDRAWAL”, “EXIT”). The function *selectedService* takes values in *Service*.

Finally it’s possible to define concrete domains that are subsets of type domains. In the ATM example the domain *MoneySize* is a concrete domain, subset of Integer.

Moreover it is possible to define domains through the application of some set operators to type domains. These are: ProductDomain, SequenceDomain, PowersetDomain, BagDomain and MapDomain.

In the previous text we have respected the theory of the ASM and we have called functions even the entities that, in a programming language, are called variables. In fact *moneyLeft, outMess* and *selectedService* can be used as variables.

In a programming language point of view a function is an application that takes values from a input domain and returns values in an output domain; an example of that can be the *pin* function: it takes values from the *Numcard* domain and returns Integer values.

Sometimes a function takes more than a value as input; in this case the input domain must be defined as a Product domain of the type domains concerned. In the LIFT example (section 8.2) the function

```
dynamic controlled hasToDeliverAt: Prod(Lift, Floor) -> Boolean
```

, for each pair ($l in Lift, $f in Floor), is true if someone at lift $l has selected the floor $f as destination.

Sometimes it could be useful to define a list (or array) of elements; this can be done through a sequence domain. Let’s see in code 2 the model that generate a sequence of Fibonacci numbers.

```
asm fibonacci

import ../STDL/StandardLibrary

signature:
    dynamic monitored size: Integer //number of Fibonacci numbers
    dynamic controlled num_fibo: Integer
    dynamic controlled index: Integer
    dynamic controlled succ_fibo: Seq(Integer)

definitions:

    macro rule r_insert_number =
        if(num_fibo=2) then
            num_fibo := size
        endif

    macro rule r_fibonacci =
        if(index<num_fibo) then
            seq
                succ_fibo := append(succ_fibo,at(succ_fibo,iton(index-1))+
                                              at(succ_fibo,iton(index-2)))
                indice := indice+1
```

```
            endseq
        endif

    main rule r_Main =
        seq
            r_insert_number[]
            r_fibonacci[]
        endseq


default init s0:
        function succ_fibo = [1, 1]
        function index = 2
        function num_fibo = 2
```

Code 2: Fibonacci sequence

The domain of *succ_fibo* is *Seq(Integer)*; all the essential function to manipulate a sequence are defined in the Standard Library [5]. In the example we use the *at* operator to select the last two elements and the *append* operator to add the new element to the sequence.

3 Body and main rule

3.1 Static domains, variables and functions initialization

In the body section the user must set the values of static concrete domains and static functions. In the ATM example (section 8.1) we can see that, at the beginning of the definitions section, we have defined the values of *MoneySize* domain (10, 20, 40, 50, 100, 150, 200). In the same example we have also set the values of the static 'variables' *minMoney* and *maxWithdrawal* that are constants that represent the minimal amount of money that the ATM machine must always contain and the maximum amount that a user can withdraw.

Then we can see the definition of a static function; a static function is used when there is a fixed connection between the elements of domain and codomain. In the ATM example the function *pin* associates each card to its pin number.

For completeness let's see the definition of a static function that has two parameters (code 3). The example is taken from [4] where is described "*the control of a package router to sort packages into bins according to their destinations*". The static function *dir* (code 3) implements a routing table for the switches, elements that the computer can flip to route a package to its destination bin.

```
function dir($sw in SwitchBin, $b in Bin) =
    switch($sw, $b)
        case (switch0_1, bin1): LEFT
        case (switch0_1, bin2): LEFT
        case (switch0_1, bin3): RIGHT
        case (switch0_1, bin4): RIGHT
        case (switch1_1, bin1): LEFT
        case (switch1_1, bin2): RIGHT
        case (switch1_1, bin3): NONE
        case (switch1_1, bin4): NONE
```

```
        case (switch1_2, bin1): NONE
        case (switch1_2, bin2): NONE
        case (switch1_2, bin3): LEFT
        case (switch1_2, bin4): RIGHT
    endswitch
```

Code 3: Package router control: static *dir* function

In this case the switch operator has been applied to a couple of elements; in general it can be applied to any n-tuple of elements.

3.2 Derived functions

In the body section, also the derived functions must be defined. A derived function is a function whose return value is subjected to its inputs. In the ATM example the function

```
derived allowed: Prod(NumCard, Integer) -> Boolean
```

return true if the balance of the card $c is greater than or equal to the requested amount of money $m. We can observe two derived functions also in the LIFT example: *canContinue* and *attracted*.

3.3 Rules

After the definition of domains and functions, the rules definition can be inserted. Each function name must start with the string *r_*. The order in which the rules are defined is very important; for example, if a rule *r_b*[] calls a rule *r_a*[], *r_a*[] must be defined before *r_b*[]. This means that, by now, it's not possible to implement an indirect recursion between rules (*r_a*[] calls *r_b*[] and *r_b*[] calls *r_a*[]).

There are two types of rule, macro and turbo. A macro rule must be called with squared brackets; a turbo one, instead, needs round brackets.

In all the examples encountered previously, we have always seen only macro rules. Let's now see in code 4 the use of a turbo rule.

```
asm recursive_factorial

import ../STDL/StandardLibrary

signature:
    dynamic controlled value: Integer

definitions:

    turbo rule r_factorial($n in Integer) in Integer =
        local x : Integer [ x:=1 ]
            if($n=1) then
                result := 1
            else
                seq
                    x <- r_factorial($n-1)
                    result := $n*x
                endseq
            endif
```

```
    main rule r_Main =
        r_factorial(value)


default init s0:
        function value = 5
```

Code 4: Example of turbo rule: Recursive factorial

In the example the rule *r_factorial* is a TurboReturnRule; the type of return value is defined in the rule signature (in this case "in Integer"). The return value of the rule is the standard variable *result*.

We can notice that in this example a local variable *x* has been used.

3.4 Main rule

After the body section, the main rule can be inserted. In general the structure of the main rule depends on the ASM specification we are writing and, also, on our "programming style". Nonetheless, in our experience, we have seen that often it is possible to structure our code in order to reproduce the different states of the ASM and the constraints that guard the passage from a state to another.

We have used two different approaches; in the first one, the structure of the finite state machine is visible. In the LIFT example (section 8.2) a *r_FSM* rule simulates the passage from a state to another: it takes as parameters the starting state ($s1), the guard of the transition ($cond), the rule to execute if the condition is true ($rule) and the final state ($s2). In the main rule all the possible transitions are evaluated in parallel.

In the second approach, instead, the structure of the FSM disappear; the guards and the state transitions are included in the rules. An example of this approach is the ATM example (section 8.1). Let's see for example the *r_insertcard*[] rule. The first instruction checks if the ATM is in the AWAITCARD state; in this case, if the $c card inserted is a valid card, the ATM can pass in the AWAITPIN state. Using this approach, in the main rule, the rules themselves (*r_insertcard*[], r*_enterPin*[], *r_chooseService*[], *r_chooseAmount*[], *r_withdrawal*[]) are evaluated in parallel.

4 Initialization

In the initialization section we can initialize the controlled functions. The techniques are the same that we have seen for static functions in section 3.1. If the initial value of a function is not specified, all its locations take the *undef* value.
Although the parser does not force you to initialize all the controlled functions, we strongly suggest to do it because, often, a missing initialization can cause a run-time error[3].

[3]The simulator, in some situations, is not able to deal with the *undef* value.

5 Monitored function

An ASM machine can communicate with the environment (the user) through the *monitored* functions. The values of the monitored functions can be passed to the *AsmetaS* simulator [2] in an interactive way (command-line) or by setting an environment file (file.env). What mode is recommended? Usually it's useful to execute a first run of our model in the interactive way, to discover in what order the monitored functions are requested; in fact it could be difficult to predict the correct order when the code is particularly complex. In code 1 the first three transaction of the ATM specification are shown. The simulator asks the user for the value of *insertedCard*, *insertedPin* and *selectedService*.

In code 5 the environment file that reproduces the same execution shown in figure 1 is shown. The # symbol is used to introduce comment lines; the file could contain only three lines with the values 'card1', '1' and 'BALANCE'. The commented lines are not essential; they have been inserted to make the file more readable to the user.

5.1 At what state the monitored functions values belong?

In this section we want to describe exactly how an ASM state is built and, in particular, how and when the monitored functions contribute to the building of the ASM state.
Let's see figure 1 in which the output of an AsmetaL model simulation is shown; you can see that the information about the function values in state i is divided in two part:

- <*State* i *(controlled)*>...</*State* i *(controlled)*> contains the values of the controlled functions;
- <*State* i *(monitored)*>...</*State* i *(monitored)*> contains the values of the monitored functions.

The information is divided in such a way because the values of the controlled functions and of the monitored functions are obtained in two different moments and it is more convenient to show them separately.

Let's see, through the observation of the simulation output, how the ASM state evolves during the simulation:

1. in the initial state the values of the controlled functions are set; you can imagine that the initial state contributes to build <*State 0 (controlled)*>;
2. in order to complete the state 0, the values of the monitored functions are needed: these are requested to the user and are reported in the console in the <*State 0 (monitored)*>[4]; in this example the simulator reports that the monitored function *insertedCard* has value *card1*;

[4]We will see in section 5.2 that, in a given state, just the values of monitored functions that can be read are requested; for this reason, some monitored functions could not be shown in some states.

3. at this point, the state 0 is complete and the simulator can calculate the update set; the update set is applied and the controlled part of state 1 (<*State 0 (controlled)*>) can be shown;

4. now the simulator asks again for monitored functions values and so on ...

It is important to notice that at the i^{th} step, after having read the monitored functions values, the simulator prints the monitored functions values of state $i-1$ and the controlled functions values of state i; indeed, the monitoring functions values permit to complete the previous state and from that calculate the update set that gives the controlled part of the new state.

5.2 How to use monitored functions

The theory of ASM says that, in any state, the values of all monitored functions must be determined; nonetheless it's important to notice that, in some states, a rule cannot fire independently from the values of the monitored functions that belong to its scope. For this reason, the simulator asks the user for the value of a monitored function only when needed (lazy evaluation). So, in writing an AsmetaL model, it's useful to neste as much as possible the use of monitored functions in order to minimize the number of requests of the simulator. In the following example:

```
signature:
    enum domain State={A, B}
    monitored function continue: boolean

definitions:
    r_stateA[] =
        skip

    rule r_main =
        if(State=A and continue=true) then
            r_stateA[]
        endif
```

Code 5: Example of monitored function

the simulator asks for the value of *continue* even if the value of *State* is not A.
The main rule can be re-written in the following way:

```
    rule r_main =
        if(State=A) then
            if(continue=true) then
                r_stateA[]
            endif
        endif
```

Code 6: Main rule rewritten

Obviously the two versions of the main rule are equivalent but, in the latter format, the simulator asks for the value of *continue* only when the function *State* is A.

In many situations this trick can be easily applied.

5.3 How to identify monitored functions

Let's see, thanks to a case study, how to identify the monitored functions of a system and the problems that could influence our choice.

First of all, it's important to remember that, through the monitored functions, the user says to the simulator that something happened in the environment, but that he should not have to worry about the duration of the effect produced by the monitored functions values. In other words, the user should not have to confirm the value of a monitored function whose value remains unchanged for more than a transition.

A good example of that is provided by the well known Lift study case [1]; in this example the functions *existsCallFromTo(Floor, dir)* and *hasToDeliverAt(Lift, Floor)* are boolean functions that, respectively, indicate if there is a call from a floor (to go up or down) and if there is call from a user inside the lift to go to a floor. These functions are defined as shared because they can be set to true by the user, when he calls the lift, and set to false by the lift when he reaches the selected floor. Since, at the moment, the simulator does not support shared functions, we have to define these two functions as monitored: this introduce a problem. For the sake of simplicity we analyse the case with one lift.

These functions values must be set in a given state and have to be kept until the lift has reached the selected floor. The problem is that, in any state, the simulator asks for the value of monitored functions: since we could not know (and we don't want to) if the lift has reached the floor, we argue that, probably is not a good idea to define *existsCallFromTo* and *hasToDeliverAt* as monitored functions. So, we have declared *existsCallFromTo* and *hasToDeliverAt* as controlled function and introduced two new monitored functions *insideCall*(Lift, Floor) and *outsideCall*(Floor, Dir); *insideCall*($l, $f) must be set to true if, in a state, a user inside the lift $l has selected the floor $f. In the same way, *outsideCall*($f, $d) must be set to true if, in a state, a user at floor $f has pressed the button to go in $d direction. *existsCallFromTo* and *hasToDeliverAt* are modified according to the values of the two monitored functions. It is important to notice that the monitored functions must be set to true only when the call happens. If there is no call, the monitored functions must be set to false. It's clear that the number of monitored functions does not change in the two versions, but in the latter one their use is much more easy: in fact the user must set a monitored function only when he wants to make a call, but he does not have to worry about the duration of the effects of his call (the number of transitions requested by the lift to reach the floor selected).

6 Invariants

In the ASM theory, invariants are used to express constraints over functions and rules. In writing an AsmetaL model the use of invariants can be useful to discover modeling errors. In general, the absence of invariants violations can not be considered as a proof of the correctness of a specification, but a violation of an invariant,

instead, is a proof of its wrongness.

6.1 Invariants declaration

Invariants must be declared just before the main rule; each invariant must be declared through the use of the keyword *invariant* and associated to a name *invariant_name*[5]. An invariant expresses a constraint over some functions and rules of the specification: these must be listed after the keyword *over*. In code 7 *term* indicates the boolean term which expresses the constraint.

```
invariant invariant_name over id_function,...,id_rule : term
```

Code 7: Structure of an invariant

6.2 Invariants violation

In code 8 we can see an example of use of invariants.

```
asm invariant_example

import ../../../STDL/StandardLibrary

signature:
  dynamic controlled fooA: Integer
  dynamic controlled fooB: Integer
  dynamic monitored monA: Boolean
  dynamic monitored monB: Boolean

definitions:

  macro rule r_a =
    if(monA) then
      fooA := fooA + 1
    endif

  macro rule r_b =
    if(monB) then
      fooB := fooB + 1
    endif

  invariant over fooA, fooB: fooA != fooB

  main rule r_main =
    par
      r_a[]
      r_b[]
    endpar

default init s0:
  function fooA = 1
  function fooB = 0
```

Code 8: Specification with invariant

[5] An invariant name must start with the string *inv_*

The invariant says that the functions *fooA* and *fooB* cannot be equal. In code 9 we can see the execution of the specification shown in code 8 and the violation of the invariant.

```
Insert a boolean constant for monA:
true
Insert a boolean constant for monB:
true
<State 0 (monitored)>
monA=true
monB=true
</State 0 (monitored)>
<State 1 (controlled)>
fooA=2
fooB=1
</State 1 (controlled)>
Insert a boolean constant for monA:
false
Insert a boolean constant for monB:
true
<State 1 (monitored)>
monA=false
monB=true
</State 1 (monitored)>
<State 2 (controlled)>
fooA=2
fooB=2
</State 2 (controlled)>
<Invariant violation>
neq(fooA,fooB)
</Invariant violation>
Final state:
fooA=2
fooB=2
```

Code 9: Invariant violation

In the initial state the invariant is satisfied: indeed *fooA* and *fooB* are different (1 and 0).
Also in the first transition the invariant has been satisfied: *fooA* is 2 and *fooB* is 1.

In the second transition, instead, the two functions assume the same value (2); the invariant has not been satisfied and the execution has been stopped. In this case the violation of the invariant has been caused by a wrong setting of monitored functions by the user.

7 Inconsistent updates

When we write an ASM specification (and so the corresponding AsmetaL code) we usually try to avoid the occurrence of an inconsistent update during the execution of the code. The parser can discover only trivial inconsistent updates (for example a function whose value is modified by two parallel instruction in the same rule); the other inconsistent updates will occur at run-time. If the inconsistent update is caused by a bad code, this can be simply resolved by a good programmer.
Sometimes, instead, inconsistent updates can be caused by the values that the user

assigns to monitored functions; we would like that the user could always set any values he wants to monitored functions but, sometimes, this is not possible. In code 10 is shown a possible code for the dining philosophers problem [6].

```
asm philosophers

import ../STDL/StandardLibrary

signature:
    domain Philosophers subsetof Agent
    abstract domain Fork
    monitored hungry : Philosophers -> Boolean
    controlled eating : Philosophers -> Boolean
    static right_fork : Philosophers -> Fork
    static left_fork : Philosophers -> Fork
    controlled owner : Fork -> Philosophers
    static phil_1: Philosophers
    static phil_2: Philosophers
    static phil_3: Philosophers
    static phil_4: Philosophers
    static phil_5: Philosophers
    static fork_1: Fork
    static fork_2: Fork
    static fork_3: Fork
    static fork_4: Fork
    static fork_5: Fork

definitions:

    function right_fork($a in Philosophers) =
        if $a = phil_1 then fork_2
        else if $a = phil_2 then fork_3
        else if $a = phil_3 then fork_4
        else if $a = phil_4 then fork_5
        else if $a = phil_5 then fork_1
        endif endif endif endif endif

    function left_fork($a in Philosophers) =
        if $a = phil_1 then fork_1
        else if $a = phil_2 then fork_2
        else if $a = phil_3 then fork_3
        else if $a = phil_4 then fork_4
        else if $a = phil_5 then fork_5
        endif endif endif endif endif


    macro rule r_Eat =
        if (hungry(self)) then
            if ( isUndef(owner(left_fork(self))) and
                 isUndef(owner(right_fork(self))) ) then
                par
                    owner(left_fork(self)) := self
                    owner(right_fork(self)) := self
                    eating(self) := true
                endpar
            endif
        endif

    macro rule r_Think =
        if ( not hungry(self)) then
            if (eating(self) and owner(left_fork(self)) = self and
                                 owner(right_fork(self)) = self )then
```

```
                par
                    owner(left_fork(self)) := undef
                    owner(right_fork(self)) := undef
                    eating(self) := false
                endpar
            endif
        endif

    main rule r_choose_philo =
        forall $p in Philosophers with true do program($p)

default init initial_state:
    function eating ($p in Philosophers)= false
    function owner ($f in Fork) = undef

    agent Philosophers :
        par
            r_Eat[]
            r_Think[]
        endpar
```

Code 10: Five philosophers

The users must set the values of the monitored function *hungry*($p in Philosophers) (i.e. the values of the function locations) to identify the philosophers that want to eat. This model makes the assumption that the user will respect the concurrency on the use of the forks and, so, that will never let two neighbour philosophers eat at the same time. In fact, the following execution causes an inconsistent update since *phil_4* and *phil_5* cannot eat at the same time:

```
INFO - <Run>
INFO - <Transition>
Insert a boolean constant for hungry(phil_4):
true
Insert a boolean constant for hungry(phil_5):
true
Exception in thread "main" org.asmeta.interpreter.UpdateClashException
```

Code 11: Five philosophers

8 Examples

8.1 ATM specification

```
asm ATM

import ../../STDL/StandardLibrary

signature:
    abstract domain NumCard
    enum domain State = { AWAITCARD | AWAITPIN | CHOOSE | OUTOFSERVICE |
    CHOOSEAMOUNT | STANDARDAMOUNTSELECTION | OTHERAMOUNTSELECTION}
    domain MoneySize subsetof Integer //tagli selezionabili
    enum domain Service = {BALANCE | WITHDRAWAL | EXIT}
    enum domain MoneySizeSelection = {STANDARD | OTHER}

    dynamic controlled currCard: NumCard
    dynamic controlled atmState: State
    dynamic controlled outMess: Any
    static pin: NumCard -> Integer
    dynamic controlled accessible: NumCard -> Boolean
    dynamic controlled moneyLeft: Integer
    dynamic controlled balance: NumCard -> Integer

    dynamic controlled numOfBalanceChecks: Integer

    dynamic monitored insertedCard: NumCard
    dynamic monitored insertedPin: Integer
    dynamic monitored selectedService: Service
    dynamic monitored insertMoneySizeStandard: MoneySize
    dynamic monitored insertMoneySizeOther: Integer
    dynamic monitored standardOrOther: MoneySizeSelection

    derived allowed: Prod(NumCard, Integer) -> Boolean

    static card1: NumCard
    static card2: NumCard
    static card3: NumCard


    static minMoney: Integer
    static maxPrelievo: Integer

definitions:
    domain MoneySize = {10, 20, 40, 50, 100, 150, 200}
    function minMoney = 200
    function maxPrelievo = 1000

    function pin($c in NumCard) =
        switch($c)
            case card1 : 1
            case card2 : 2
            case card3 : 3
        endswitch

    function allowed($c in NumCard, $m in Integer) =
        balance($c) >= $m

    macro rule r_subtractFrom ($c in NumCard, $m in Integer) =
        balance($c) := balance($c) - $m
```

```
macro rule r_goOutOfService =
    par
        atmState := OUTOFSERVICE
        outMess := "Out of Service"
    endpar

macro rule r_insertcard =
    if(atmState=AWAITCARD) then
        if(exist $c in NumCard with $c=insertedCard) then
            par
                currCard := insertedCard
                atmState := AWAITPIN
                outMess := "Enter pin"
            endpar
        endif
    endif

macro rule r_enterPin =
    if(atmState=AWAITPIN) then
        if(insertedPin=pin(currCard) and accessible(currCard)) then
            par
                outMess := "Choose service"
                atmState := CHOOSE
                numOfBalanceChecks := 0
            endpar
        else
            par
                atmState := AWAITCARD
                if(insertedPin!=pin(currCard)) then
                    outMess := "Wrong pin"
                endif
                if(not(accessible((currCard))) and insertedPin=pin(
currCard)) then
                    outMess := "Account non accessible"
                endif
            endpar
        endif
    endif

macro rule r_chooseService =
    if(atmState=CHOOSE) then
        par
            if(selectedService=BALANCE) then
                if(numOfBalanceChecks = 0) then
                    par
                        numOfBalanceChecks := numOfBalanceChecks + 1
                        outMess := balance(currCard)
                    endpar
                else
                    par
                        atmState := AWAITCARD
                        outMess := "You can check only once your
balance. Goodbye."
                    endpar
                endif
            endif
            if(selectedService=WITHDRAWAL) then
                par
                    atmState := CHOOSEAMOUNT
                    outMess := "Choose Standard or Other"
                endpar
            endif
```

```
            if(selectedService=EXIT) then
                par
                    atmState := AWAITCARD
                    outMess := "Goodbye"
                endpar
            endif
        endpar
    endif

rule r_chooseAmount =
    if(atmState=CHOOSEAMOUNT) then
        par
            if(standardOrOther=STANDARD) then
                par
                    atmState := STANDARDAMOUNTSELECTION
                    outMess := "Select a money size"
                endpar
            endif
            if(standardOrOther=OTHER) then
                par
                    atmState := OTHERAMOUNTSELECTION
                    outMess := "Enter money size"
                endpar
            endif
        endpar
    endif

rule r_grantMoney($m in Integer) =
    par
        r_subtractFrom[currCard, $m]
        moneyLeft := moneyLeft - $m
        seq
            accessible(currCard) := false
            accessible(currCard) := true
        endseq
        atmState := AWAITCARD
        outMess := "Goodbye"
    endpar

macro rule r_processMoneyRequest ($m in Integer) =
    if(allowed(currCard, $m)) then
        r_grantMoney[$m]
    else
        outMess := "Not enough money in your account"
    endif

macro rule r_prelievo =
    par
        if(atmState=STANDARDAMOUNTSELECTION) then
            if(exist $m in MoneySize with $m=insertMoneySizeStandard)
then
                if(insertMoneySizeStandard<=moneyLeft) then
                    r_processMoneyRequest [insertMoneySizeStandard]
                else
                    outMess := "Il bancomat non ha abbastanza soldi"
                endif
            endif
        endif
        if(atmState=OTHERAMOUNTSELECTION) then
            if(mod(insertMoneySizeOther, 10)=0) then
                if(insertMoneySizeOther<=moneyLeft) then
                    r_processMoneyRequest [insertMoneySizeOther]
```

```
                    else
                        outMess := "Il bancomat non ha abbastanza soldi"
                    endif
                else
                    outMess := "Tagli non compatibili"
                endif
            endif
        endpar

    main rule r_Main =
        if (moneyLeft < minMoney) then
            r_goOutOfService[]
        else
            par
                r_insertcard[]
                r_enterPin[]
                r_chooseService[]
                r_chooseAmount[]
                r_prelievo[]
            endpar
        endif

default init s0:
    function atmState = AWAITCARD
    function moneyLeft = 1000
    function balance($c in NumCard) = switch($c)
                                        case card1 : 3000
                                        case card2 : 1652
                                        case card3 : 548
                                      endswitch
    function accessible($c in NumCard) = true
```

8.2 LIFT specification

```
asm LIFT

import ../STDL/StandardLibrary

signature:
  abstract domain Lift
  enum domain Dir = {UP | DOWN}
  domain Floor subsetof Integer
  enum domain State = {HALTING | MOVING}

  dynamic controlled direction:  Lift -> Dir
  dynamic controlled ctlState: Lift -> State
  dynamic controlled floor:  Lift -> Floor

  derived attracted: Prod(Dir, Lift) -> Boolean
  derived canContinue: Prod(Dir,Lift) -> Boolean

  dynamic controlled hasToDeliverAt: Prod(Lift, Floor) -> Boolean
  dynamic controlled existsCallFromTo: Prod(Floor, Dir) -> Boolean
  dynamic monitored insideCall: Prod(Lift, Floor) -> Boolean
  dynamic monitored outsideCall: Prod(Floor, Dir) -> Boolean

  static plusorminus: Dir -> Integer
  static opposite: Dir -> Dir
  static ground: Integer
  static top: Integer

  static lift1: Lift // there is one lift

definitions:
  domain Floor = {0..+2}
  function ground = 0
  function top = 2

  function plusorminus($d in Dir) =
    if($d = UP) then
      1
    else
      -1
    endif

  function opposite ($d in Dir) =
    if ($d = UP) then
      DOWN
    else
      UP
    endif

  function canContinue($d in Dir, $l in Lift) =
     (((($d =UP) and (attracted(UP,$l) and not(hasToDeliverAt($l,floor($l))
    )
         and not(existsCallFromTo(floor($l),UP))))
      or (($d =DOWN) and (attracted(DOWN,$l) and not(hasToDeliverAt($l,
    floor($l)))
        and not(existsCallFromTo(floor($l),DOWN))))  )

  function attracted($d in Dir, $l in Lift) =
    if($d = UP) then
      if (exist $f in Floor with ($f > floor($l) and
```

```
                  (hasToDeliverAt($l,$f) or
                   existsCallFromTo($f,UP) or
                   existsCallFromTo($f,DOWN) ))) then
      true
    else
      false
    endif
  else
    if(exist $g in Floor with ($g < floor($l) and
                  (hasToDeliverAt($l,$g) or
                 existsCallFromTo($g,UP) or
                 existsCallFromTo($g,DOWN) ))) then
      true
    else
      false
    endif
  endif


macro rule r_costraintHasToDeliver =
  forall $f in Floor, $l in Lift with (hasToDeliverAt ($l,$f) and
                    ctlState($l) = HALTING and
                    floor($l) = $f) do
    hasToDeliverAt ($l,$f) := false

macro rule r_costraintCallFromTo =
  forall $f in Floor, $d in Dir, $l in Lift with (existsCallFromTo($f,$d
  ) and
                          (($f=ground and $d=DOWN) or
                           ($f=top and $d=UP) or
                           (ctlState($l)=HALTING and
                            floor($l) = $f))) do
    existsCallFromTo ($f,$d) := false

rule r_Depart($l in Lift) =
  floor($l) := floor($l) + plusorminus(direction($l))

rule r_Continue($l  in Lift) =
  floor($l) := floor($l) + plusorminus(direction($l))

macro rule r_CancelRequest($f in Floor,$d in Dir, $l in Lift) =
  par
    hasToDeliverAt($l,$f) := false
    existsCallFromTo($f,$d) := false
  endpar

rule r_Stop($l in Lift) =
  r_CancelRequest[floor($l),direction($l), $l]

rule r_Change ($l in Lift) =
  par
    direction($l) := opposite(direction($l))
    r_CancelRequest[floor($l),opposite(direction($l)), $l]
  endpar

rule r_Fsm($l in Lift, $s1 in State, $cond in Boolean,
           $rule in Rule, $s2 in State) =
  if ctlState($l) = $s1 and $cond then
    par
      $rule
      ctlState($l) := $s2
    endpar
```

```
    endif

  main rule r_Main =
    forall $l in Lift with true do
      seq
        forall $f in Floor do
          par
            if(outsideCall($f,DOWN)=true) then
              existsCallFromTo($f,DOWN):= true
            endif
            if(outsideCall($f,UP)=true) then
              existsCallFromTo($f,UP):= true
            endif
            if(insideCall($l,$f)=true) then
              hasToDeliverAt($l,$f):= true
            endif
          endpar
          r_costraintCallFromTo[]
          r_costraintHasToDeliver[]
          par
            r_Fsm[$l, HALTING, attracted(direction($l), $l),
                <<r_Depart[$l]>>, MOVING]
            r_Fsm[$l, MOVING, canContinue(direction($l),$l),
                 <<r_Continue[$l]>>, MOVING]
            r_Fsm[$l, MOVING, not canContinue(direction($l),$l),
                <<r_Stop[$l]>>, HALTING]
            r_Fsm[$l, HALTING, not attracted(direction($l), $l) and
                      attracted(opposite(direction($l)), $l),
                 <<r_Change[$l]>>, HALTING]
          endpar
      endseq


default init s0:
  function floor($l in Lift) = 0
  function direction($l in Lift) = UP
  function ctlState($l in Lift)= HALTING
  function hasToDeliverAt($l in Lift, $i in Integer) = false
  function existsCallFromTo($i in Integer, $d in Dir) = false
```

References

[1] E. Börger and R. F. Stärk. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003.

[2] ASMETA web site: http://asmeta.sourceforge.net/

[3] AsmM Concrete Syntax (AsmetaL) v.2.0.0. `http://fmlab.dti.unimi.it/asmeta/download/AsmetaL_quickguide.html`

[4] E. Börger. *The Abstract State Machines Method for High-Level System Design and Analysis.* Package Router Control.

[5] *The AsmM Standard Library*. A library of predefined ASM domains and functions. `http://fmlab.dti.unimi.it/asmeta/download/StandardLibrary.asm`

[6] The dining philosophers problem. `http://en.wikipedia.org/wiki/Dining_philosophers_problem`

```
Insert a abstract constant in NumCard for insertedCard:
card1
<State 0 (monitored)>
NumCard={card1,card2,card3}
insertedCard=card1
</State 0 (monitored)>
<State 1 (controlled)>
NumCard={card1,card2,card3}
atmState=AWAITPIN
currCard=card1
moneyLeft=1000
outMess=Enter pin
</State 1 (controlled)>
Insert a integer constant for insertedPin:
1
<State 1 (monitored)>
NumCard={card1,card2,card3}
insertedPin=1
</State 1 (monitored)>
<State 2 (controlled)>
NumCard={card1,card2,card3}
accessible(card1)=true
atmState=CHOOSE
currCard=card1
moneyLeft=1000
numOfBalanceChecks=0
outMess=Choose service
</State 2 (controlled)>
Insert a symbol of Service in [BALANCE, WITHDRAWAL, EXIT] for
    selectedService:
BALANCE
<State 2 (monitored)>
NumCard={card1,card2,card3}
selectedService=BALANCE
</State 2 (monitored)>
<State 3 (controlled)>
NumCard={card1,card2,card3}
accessible(card1)=true
atmState=CHOOSE
balance(card1)=3000
currCard=card1
moneyLeft=1000
numOfBalanceChecks=1
outMess=3000
</State 3 (controlled)>
```

Figure 1: Monitored function - ATM specification simulation result

```
#Insert a abstract constant in NumCard for insertedCard:
card1
#Insert a integer constant for insertedPin:
1
#Insert a symbol of Service in [BALANCE | WITHDRAWAL | EXIT] for
    selectedService:
BALANCE
```

Figure 2: Monitored function - Environment file